ECE-6483 Real Time Embedded Systems
Midterm

New York University, Spring 2025

Due on 03/20, 11:59 PM

Name: Raman Kumar Jha
NYU ID: N13866145

Question 1

We spent a significant amount of time in class discussing how to read and write memory using both C and assembly. Since our architecture uses memory-mapped I/O, setting up the GPIO registers is simply reading and writing the appropriate memory locations. The following algorithm (A, B, C, D) can be used for our controller to do a block GPIO write, meaning writing all port pins (15 in total) simultaneously, for any port (say PORTA). **PORTA has a base address of 0x40020000**.

A. Set each pin in the MODER register to general-purpose output mode

8.4.1 GPIO port mode register (GPIOx_MODER) (x = A..I/J/K)

Address offset: 0x00

Reset values:

- 0xA800 0000 for port A
- 0x0000 0280 for port B
- 0x0000 0000 for other ports

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

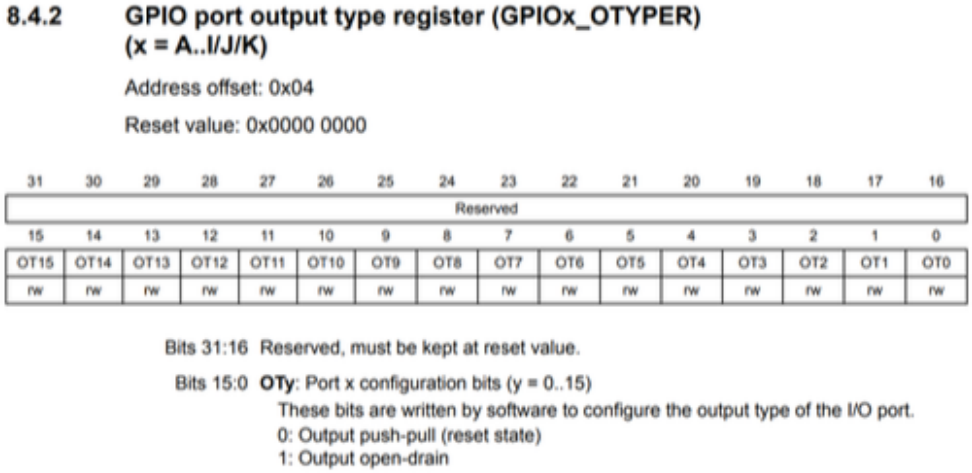
Bits 2y:2y+1 **MODERy[1:0]:** Port x configuration bits (y = 0..15)
These bits are written by software to configure the I/O direction mode.
00: Input (reset state)
01: General purpose output mode
10: Alternate function mode
11: Analog mode

The MODER register is located at offset `0x00` from the base address. Each pin requires two bits to configure its mode:

- `00`: Input (reset state)
- `01`: General-purpose output mode
- `10`: Alternate function mode
- `11`: Analog mode

To set all pins to general-purpose output mode, we need to write **01** to all pin configuration bits.

B. Set each pin in the OTYPER register to output push-pull



8.4.2 GPIO port output type register (GPIOx_OTYPER) (x = A..I/J/K)

Address offset: 0x04

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OT15	OT14	OT13	OT12	OT11	OT10	OT9	OT8	OT7	OT6	OT5	OT4	OT3	OT2	OT1	OT0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **OTy**: Port x configuration bits (y = 0..15)
These bits are written by software to configure the output type of the I/O port.
0: Output push-pull (reset state)
1: Output open-drain

The OTYPER register is located at offset `0x04` from the base address. Each pin requires one bit to configure its output type:

- `0`: Output push-pull (reset state)
- `1`: Output open-drain

To set all pins to output push-pull, we need to write **0** to all pin configuration bits.

C. Set each pin in the PUPDR register to 0 (no pull-up or pull-down resistor)

The PUPDR register is located at offset `0x0C` from the base address. Each pin requires two bits to configure its pull-up/pull-down resistor:

- `00`: No pull-up/pull-down
- `01`: Pull-up
- `10`: Pull-down
- `11`: Reserved

To disable pull-up and pull-down resistors for all pins, we need to write **00** to all pin configuration bits.

8.4.4 GPIO port pull-up/pull-down register (GPIOx_PUPDR) (x = A..I/J/K)

Address offset: 0x0C

Reset values:

- 0x6400 0000 for port A
- 0x0000 0100 for port B
- 0x0000 0000 for other ports

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PUPDR15[1:0]		PUPDR14[1:0]		PUPDR13[1:0]		PUPDR12[1:0]		PUPDR11[1:0]		PUPDR10[1:0]		PUPDR9[1:0]		PUPDR8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PUPDR7[1:0]		PUPDR6[1:0]		PUPDR5[1:0]		PUPDR4[1:0]		PUPDR3[1:0]		PUPDR2[1:0]		PUPDR1[1:0]		PUPDR0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 2y:2y+1 **PUPDRy[1:0]**: Port x configuration bits (y = 0..15)
These bits are written by software to configure the I/O pull-up or pull-down
00: No pull-up, pull-down
01: Pull-up
10: Pull-down
11: Reserved

D. Set the ODR register bits to 1 or 0 depending on whether the pin is turned on or off

8.4.6 GPIO port output data register (GPIOx_ODR) (x = A..I/J/K)

Address offset: 0x14

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **ODRy**: Port output data (y = 0..15)
These bits can be read and written by software.
Note: For atomic bit set/reset, the ODR bits can be individually set and reset by writing to the GPIOx_BSRR register (x = A..I/J/K).

a. Write a C function:

```
uint32_t  BlockWritePortA(uint16_t PinVals, uint16_t PullUP) {

//Code Here
}
```

PinVals is a 16 bit number with 1's in positions to turn "on" and 0's in the pin positions to turn "off". PullUP is a 16 bit mask, where the bit positions with a "1" will have a pull-up resistor enabled. Positions with a "0" will have no pull-up enabled. The function should implement algorithms parts A., B., C. and D. described above and return the value of the ODR register above. Note, the addresses of the registers (memory) are the Base Address + Offset as indicated above.

The ODR register is located at offset `0x14` from the base address. Each pin requires one bit:

- We need to write **1** to turn the pin on.
- We need to write **0** to turn the pin off.

Part a: Write a C Function

```
uint32_t  BlockWritePortA(uint16_t PinVals, uint16_t PullUP) {

//Code Here
}
```

PinVals is a 16 bit number with 1's in positions to turn "on" and 0's in the pin positions to turn "off". PullUP is a 16 bit mask, where the bit positions with a "1" will have a pull-up resistor enabled. Positions with a "0" will have no pull-up enabled. The function should implement algorithms parts A., B., C. and D. described above and return the value of the ODR register above. Note, the addresses of the registers (memory) are the Base Address + Offset as indicated above.

Below is the implementation of the function in C:

```c
#include <stdint.h>

uint32_t BlockWritePortA(uint16_t PinVals, uint16_t PullUP) {
    // Base address for PORTA
    volatile uint32_t *MODER = (uint32_t *)0x40020000;
    volatile uint32_t *OTYPER = (uint32_t *)0x40020004;
    volatile uint32_t *PUPDR = (uint32_t *)0x4002000C;
    volatile uint32_t *ODR = (uint32_t *)0x40020014;

    // Step A: Set MODER register for general-purpose output mode
    *MODER = 0x55555555; // Each pin gets "01" (general-purpose output)

    // Step B: Set OTYPER register for output push-pull
    *OTYPER = 0x00000000; // All pins set to "push-pull"

    // Step C: Set PUPDR register for no pull-up/pull-down resistors
    *PUPDR = PullUP; // Apply PullUP mask

    // Step D: Set ODR register based on PinVals
    *ODR = PinVals; // Write PinVals directly

    return *ODR; // Return current value of ODR register
}
```

Part b: Write an ARM Assembly Function

Below is the implementation of the function in ARM assembly:

NOTE: Be careful how you use LDR, as there are limits to the "mem" parameter!

```
.globl _BlockWritePortA

.syntax unified


_BlockWritePortA

;

; Code Here

;

bx lr
```

```
.global _BlockWritePortA
.syntax unified

_BlockWritePortA:
    // Load base address of PORTA into R2
    LDR R2, =0x40020000

    // Step A: Set MODER register for general-purpose output mode
    LDR R3, =0x55555555    // Value for MODER
    STR R3, [R2]           // Store value into MODER

    // Step B: Set OTYPER register for output push-pull
    LDR R3, =0x00000000    // Value for OTYPER
    STR R3, [R2, #4]       // Store value into OTYPER

    // Step C: Set PUPDR register based on PullUP parameter
    LDR R3, [SP, #4]       // Load PullUP parameter from stack
    STR R3, [R2, #12]      // Store value into PUPDR

    // Step D: Set ODR register based on PinVals parameter
    LDR R3, [SP]           // Load PinVals parameter from stack
    STR R3, [R2, #20]      // Store value into ODR

    BX LR                  // Return from function
```

Question 2: Consider the ARM Assembly code segment

(a) Accurately comment each line of code

```
// Function declaration
.global _MyFunc

.text
_MyFunc:
    stmfdb sp!, {r4, r5, r6, r7, r8, lr} // Save registers r4-r8 and link register (l
    cmp r1, #1                           // Compare r1 (size) with 1
    ble end_outer                        // If r1 <= 1, branch to end_outer

    sub r5, r1, #1                       // Set r5 = r1 - 1 (remaining iterations)
    mov r4, r0                           // Set r4 = r0 (pointer to data array)
    mov r6, #0                           // Initialize flag (r6) to 0

loop_start:
    ldr r7, [r4], #4                     // Load a word from memory pointed by r4 to r
    ldr r8, [r4]                         // Load another word from memory pointed by r
    cmp r7, r8                           // Compare values in r7 and r8
    ble no_go                            // If r7 <= r8, skip swapping

    mov r6, #1                           // Set flag (r6) to 1 indicating a swap occur
    sub r4, r4, #4                       // Restore pointer to previous position
    swp r8, r8, [r4]                     // Swap value in memory at [r4] with value in
    str r8, [r4], #4                     // Store updated value back in memory and inc

no_go:
    subs r5, r5, #1                      // Decrement loop counter (r5)
    bne loop_start                       // If counter != 0, repeat loop

end_inner:
    cmp r6, #0                           // Check if any swaps occurred
    beq end_outer                        // If no swaps occurred (flag == 0), exit out

    mov r6, #0                           // Reset flag for next pass
    mov r4, r0                           // Reset pointer to start of array
    sub r5, r1, #1                       // Reset loop counter for next pass
    b loop_start                         // Restart inner loop

end_outer:
    ldmfd sp!, {r4, r5, r6, r7, r8, pc} // Restore registers and return from function
```

(b) Describe what parameters `r0` and `r1` are used for.

- **r0**: Pointer to the data array that needs to be processed.

- **r1**: Size of the data array.

(c) What are each of the local variables `r4-r8` used for?

- **r4**: Pointer to the current position in the data array.
- **r5**: Loop counter for the number of iterations.
- **r6**: Flag indicating whether a swap occurred during an iteration.
- **r7**: Temporary storage for a word loaded from memory.
- **r8**: Temporary storage for another word loaded from memory or used during swapping.

(d) What is the purpose of this function?

The purpose of this function is to perform a bubble sort on an array of integers. It iteratively compares adjacent elements in the array and swaps them if they are out of order. The process repeats until no swaps are needed during a pass through the array.

(e) Explain in detail the specific purpose of `stmfd` and `ldmfd` in this function.

stmfd: This instruction is used at the start of the function to save registers (**r4-r8**) and the link register (**lr**) onto the stack. The **-fd** suffix means "Full Descending," which indicates that the stack grows downward and stores data starting from one address below the current stack pointer (**sp**). This ensures that these registers' values are preserved across function calls.

ldmfd: This instruction is used at the end of the function to restore previously saved registers (**r4-r8**) and return control to the caller by restoring the program counter (**pc**). The stack pointer (**sp**) is adjusted upward as part of this operation.

Together, these instructions ensure proper preservation and restoration of register values across function calls.

Question 3

Suppose you want to use your microcontroller to perform complex math, like a DFT for example, which requires a way to store and manipulate complex numbers.

Part (a): Design a global array variable that is capable of storing 20 complex numbers in terms of their real and imaginary parts. You can use any data types, including structs, to store the array of 20 complex numbers.

We use a `struct` to represent a complex number with real and imaginary parts. The global array can store 20 such complex numbers.

C Code:

```
typedef struct {
    float real;
    float imag;
} Complex;

Complex complexArray[20];
```

Part (b): Write a C function called: retrieveValue(x,n) //returns a pointer to the "n th" complex number in the array //stored at address x

The function returns a pointer to the n th complex number in the array which is stored at address x.

C Code:

```
Complex* retrieveValue(Complex* x, int n) {
    return &x[n];
}
```

Part (c): Write the same function in ARM assembly

Below is the ARM assembly implementation of the `retrieveValue` function.

ARM Assembly Code:

```
retrieveValue:
    LSL r1, r1, #3        // Multiply n by 8 (size of Complex struct: 2 floats, each
    ADD r0, r0, r1        // Add offset to base address x
    BX lr                 // Return pointer to n-th complex number
```